Adaptive Insertion Policies for High Performance Caching

Moinuddin K. Qureshi† Aamer Jaleel§ Yale N. Patt† Simon C. Steely Jr.§ Joel Emer§

†ECE Department
The University of Texas at Austin
{moin, patt}@hps.utexas.edu

§Intel Corporation, VSSAD
Hudson, MA
{aamer.jaleel, simon.c.steely.jr, joel.emer}@intel.com

ABSTRACT

The commonly used LRU replacement policy is susceptible to thrashing for memory-intensive workloads that have a working set greater than the available cache size. For such applications, the majority of lines traverse from the MRU position to the LRU position without receiving any cache hits, resulting in inefficient use of cache space. Cache performance can be improved if some fraction of the working set is retained in the cache so that at least that fraction of the working set can contribute to cache hits.

We show that simple changes to the *insertion policy* can significantly reduce cache misses for memory-intensive workloads. We propose the *LRU Insertion Policy (LIP)* which *places* the incoming line in the LRU position instead of the MRU position. LIP protects the cache from thrashing and results in close to optimal hit-rate for applications that have a cyclic reference pattern. We also propose the *Bimodal Insertion Policy (BIP)* as an enhancement of LIP that adapts to changes in the working set while maintaining the thrashing protection of LIP. We finally propose a *Dynamic Insertion Policy (DIP)* to choose between BIP and the traditional LRU policy depending on which policy incurs fewer misses. The proposed insertion policies do not require any change to the existing cache structure, are trivial to implement, and have a storage requirement of less than two bytes. We show that DIP reduces the average MPKI of the baseline 1MB 16-way L2 cache by 21%, bridging two-thirds of the gap between LRU and OPT.

Categories and Subject Descriptors:
B.3.2 [Design Styles]: Cache memories

General Terms: Design, Performance.

Keywords: Replacement, Thrashing, Set Sampling, Set Dueling.

1. INTRODUCTION

The LRU replacement policy and its approximations have remained as the de-facto standard for replacement policy in on-chip caches over the last several decades. While the LRU policy has the advantage of good performance for high-locality workloads, it can have a pathological behavior for memory-intensive workloads that have a working set greater than the available cache size. There have been numerous proposals to improve the performance of LRU, however, many of these proposals incur a huge storage overhead, significant changes to the existing design, and poor performance for LRU-friendly workloads. Every added structure and change to the existing design requires design effort, verification effort, and testing effort. Therefore, it is desirable that changes to the conventional replacement policy require minimal changes to the existing design, require no additional hardware structures, and perform well for a wide variety of applications. This paper focuses on designing a cache replacement policy that performs well for both LRU-friendly and LRU-averse workloads while requiring negligible hardware overhead and changes.

We divide the problem of cache replacement into two parts: *victim selection policy* and *insertion policy*. The victim selection policy decides which line gets evicted for storing an incoming line, whereas, the insertion policy decides where in the replacement list the incoming line is placed. For example, the traditional LRU replacement policy inserts the incoming line in the MRU position, thus using the policy of *MRU Insertion*. Inserting the line in the MRU position gives the line a chance to obtain a hit while it traverses all the way from the MRU position to the LRU position. While this may be a good strategy for workloads whose working-set is smaller than the available cache size or for workloads that have high temporal locality, such an insertion policy causes thrashing for memory-intensive workloads that have a working set greater than the available cache size. We show that with the traditional LRU policy, more than 60% of the lines installed in the L2 cache remain unused between insertion and eviction. Thus, most of the inserted lines occupy cache space without ever contributing to cache hits. When the working set is larger than the available cache size, cache performance can be improved by retaining some fraction of the working set long enough that at least that fraction of the working set contributes to cache hits. However, the traditional LRU policy offers no protection for retaining the cache lines longer than the cache capacity.

We show that simple changes to the insertion policy can significantly improve cache performance for memory-intensive workloads while requiring negligible hardware overhead. We propose the *LRU Insertion Policy (LIP)* which places *all* incoming lines in the LRU position. These lines are promoted from the LRU position to the MRU position only if they get referenced while in the LRU position. LIP prevents thrashing for workloads whose working set is greater than the cache size and obtains near-optimal hit rates for workloads that have a cyclic access pattern. LIP can easily be implemented by avoiding the recency update at insertion.

LIP may retain the lines in the non-LRU position of the recency stack even if they cease to contribute to cache hits. Since LIP does not have an aging mechanism, it may not respond to changes in the working set of a given application. We propose the *Bimodal Insertion Policy (BIP)*, which is similar to LIP, except that BIP infrequently (with a low probability) places the incoming line in the MRU position. We show that BIP adapts to changes in the working set while retaining the thrashing protection of LIP.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'07, June 9–13, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-706-3/07/0006 ...$5.00.

For LRU-friendly workloads that favor the traditional policy of MRU insertion, the changes to the insertion policy are detrimental to cache performance. We propose a *Dynamic Insertion Policy (DIP)* to choose between the traditional LRU policy and BIP depending on which policy incurs fewer misses. DIP requires runtime estimates of misses incurred by each of the competing policies. To implement DIP without requiring significant hardware overhead, we propose *Set Dueling*. The Set Dueling mechanism dedicates a few sets of the cache to each of the two competing policies and uses the policy that performs better on the *dedicated sets* for the remaining *follower sets*. We analyze both analytical as well as empirical bounds for the number of dedicated sets and show that as few as 32 to 64 dedicated sets are sufficient for Set Dueling to choose the best policy. An implementation of DIP using Set Dueling requires no extra storage other than a single saturating counter and performs similar to LRU for LRU-friendly workloads.

Insertion policies come into effect only during cache misses, therefore, changes to the insertion policy do not affect the access time of the cache. The proposed changes to the insertion policy are particularly attractive as they do not require *any* changes to the structure of an existing cache design, incur only a negligible amount of logic circuitry, and have a storage overhead of less than two bytes. Our evaluations, with 16 memory-intensive benchmarks, show that DIP reduces the average misses per 1000 instructions (MPKI) for a 1MB 16-way LRU-managed L2 cache by 21.3%, bridging two-thirds of the gap between LRU and Belady's Optimal replacement (OPT) [1].

2. MOTIVATION

A miss in the L2 cache (last-level cache in our studies) stalls the processor for hundreds of cycles, therefore, our study is focused on reducing L2 misses by managing the L2 cache efficiently. The access stream visible to the L2 cache has filtered temporal locality due to the hits in the first-level cache. The loss of temporal locality causes a significant percentage of L2 cache lines to remain unused. We refer to cache lines that are not referenced between insertion and eviction as *zero reuse lines*. Figure 1 shows that for the baseline 1MB 16-way LRU-managed L2 cache, *more than half the lines installed in the cache are never reused before getting evicted.* Thus, the traditional LRU policy results in inefficient use of cache space as most of the lines installed occupy cache space without contributing to cache hits.

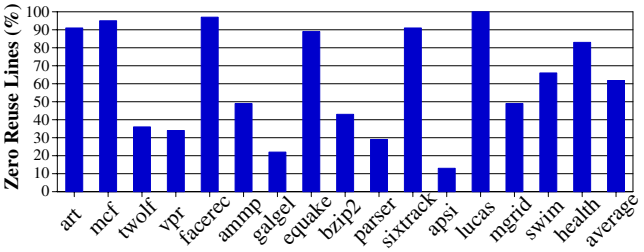


Figure 1: Zero Reuse Lines for 1MB 16-way L2 cache

Zero reuse lines occur because of two reasons. First, the line has no temporal locality which means that the line is never re-referenced. It is not beneficial to insert such lines in the cache. Second, the line is re-referenced at a distance greater than the cache size, which causes the LRU policy to evict the line before it gets reused. Several studies have investigated bypassing [10][3][6][19] and early eviction [21][20] of lines with poor temporal locality. However, temporal locality exploited by the cache is a function of both the replacement policy and the size of the working set relative to the available cache size. For example, if a workload frequently reuses a working set of 2 MB, and the available cache size is 1MB, then the LRU policy will cause all the installed lines to have poor temporal locality. In such a case, bypassing or early evicting all the lines in the working set will not improve cache performance. The optimal policy in such cases is to retain some fraction of the working set long-enough so that at least that fraction of the working set provides cache hits. However, the traditional LRU policy offers no protection for retaining the cache lines longer than the cache capacity.

For workloads with a working set greater than the available cache size, cache performance can be significantly improved if the cache can retain some fraction of the working set. To achieve this, we separate the replacement policy into two parts: *victim selection policy* and *insertion policy*. The victim selection policy decides which line gets evicted for storing an incoming line. The insertion policy decides where in the replacement list the incoming line is placed. We propose simple changes to the insertion policy that significantly improve cache performance of memory-intensive workloads while requiring negligible overhead. We present our methodology before discussing the solution.

3. EXPERIMENTAL METHODOLOGY

3.1 Configuration

We use a trace-driven cache simulator for all the experiments in the paper, except for the IPC results shown in Section 6.3. We defer the description of our execution driven simulator to that section. Table 1 shows the parameters of the first level instruction (I) and data (D) caches that we used to generate the traces for our second level cache. The L1 cache parameters were kept constant for all experiments. The baseline L2 cache is 1MB 16-way set associative. All caches in the baseline use a 64B line-size. We do not enforce inclusion in our memory model. Throughout this paper, *LRU* denotes the traditional LRU policy which inserts all incoming lines in the MRU position. Unless stated otherwise, all caches use the LRU policy for replacement decisions.

Table 1: Cache Configuration

L1 I-Cache	16kB; 64B linesize; 2-way with LRU repl.
L1 D-Cache	16kB; 64B linesize; 2-way with LRU repl.
Baseline L2	1 MB; 64B linesize; 16-way with LRU repl.

3.2 Benchmarks

The SPEC CPU2000 benchmarks used in our study were compiled for the Alpha ISA with `-fast` optimizations and profiling feedback enabled. For each SPEC benchmark, we use a representative sample of 250M instructions obtained with a tool that we developed using the SimPoint [12] methodology. Since cache replacement does not affect the number of compulsory misses, benchmarks that have a high percentage of compulsory misses are unlikely to benefit from improvements in cache replacement algorithms. Therefore, we show detailed results only for benchmarks for which approximately 50% or fewer misses are compulsory misses.[1] In addition to the SPEC benchmarks, we also used the health benchmark from the Olden suite as it represents a workload in which the working set increases with time. We ran the health benchmark to completion. Table 2 shows the fast-forward interval (FFWD), the number of L2 misses per 1000 instructions (MPKI), and the percentage of misses that are compulsory misses for each benchmark.

[1]For the 11 SPEC benchmarks excluded from our studies, the proposed technique (DIP) changes MPKI by ≤ 0.01.

Table 2: Benchmark summary (B = Billion)

Name	FFWD	MPKI	Compulsory Misses
art	18.25B	38.7	0.5%
mcf	14.75B	136	1.8%
twolf	30.75B	3.48	2.9%
vpr	60B	2.16	4.3%
facerec	111.75B	3.66	4.8%
ammp	4.75B	2.83	5.0%
galgel	14B	5.34	5.9%
equake	26.25B	18.4	14.2%
bzip2	2.25B	2.4	14.8%
parser	66.25B	1.57	20.0%
sixtrack	8.5B	0.42	20.7%
apsi	3.25B	0.32	21.4%
lucas	2.5B	16.2	41.6%
mgrid	3.5B	7.73	46.6%
swim	3.5B	23.0	50.4%
health	0B	61.7	0.73%

4. STATIC INSERTION POLICIES

The traditional LRU replacement policy inserts all incoming lines in the MRU position. Inserting the line in the MRU position gives the line a chance to obtain a hit while it traverses all the way from the MRU position to the LRU position. While this may be a good strategy for workloads whose working set is smaller than the available cache size or for workloads that have high temporal locality, such an insertion policy causes thrashing for memory-intensive workloads that have a working set greater than the available cache size. When the working set is greater than the available cache size, cache performance can be improved by retaining some fraction of the working set long enough that at least that fraction of the working set results in a cache hits.

For such workloads, we propose the *LRU Insertion Policy (LIP)*, which places *all* incoming lines in the LRU position. These lines are promoted from the LRU position to the MRU position only if they are reused while in the LRU position. LIP prevents thrashing for workloads that reuse a working set greater than the available cache size. To our knowledge this is the first study to investigate the insertion of *demand* lines in the LRU position. Earlier studies [2] have proposed to insert prefetched lines in the LRU position to reduce the pollution caused by inaccurate prefetching. However, they were targeting the problem of extraneous references generated by the prefetcher while our study is targeted towards the fundamental locality problem in memory reference streams. With their proposal, demand lines are still inserted in the MRU position making the cache susceptible to thrashing by demand references. Our proposal, LIP, protects the cache from thrashing by inserting *all* incoming lines in the LRU position. Our work can be combined with Lin et al.'s work to protect the cache from both thrashing as well as prefetcher pollution.

LIP may retain the lines in the non-LRU position of the recency stack even if they cease to be re-referenced. Since LIP does not have an aging mechanism, it may not respond to changes in the working set of the given application. We propose the *Bimodal Insertion Policy (BIP)* which is similar to LIP, except that it infrequently (with a low probability) places some incoming lines into the MRU position. BIP is regulated by a parameter, *bimodal throttle parameter* (ϵ), which controls the percentage of incoming lines that are placed in the MRU position. Both the traditional LRU policy and LIP can be viewed as a special case of BIP with $\epsilon = 1$ and $\epsilon = 0$ respectively. In Section 4.1 we show that for small values of ϵ, BIP can adapt to changes in the working set while retaining the thrashing protection of LIP.

4.1 Analysis with Cyclic Reference Model

To analyze workloads that cause thrashing with the LRU policy, we use a theoretical model of cyclic references. A similar model has been used earlier by McFarling [10] for modeling conflict misses in a direct-mapped instruction cache. Let a_i denote the address of a cache line. Let $(a_1 \cdots a_T)$ denote a temporal sequence of references a_1, a_2, ..., a_T. A temporal sequence that repeats for N times is represented as $(a_1 \cdots a_T)^N$.

Let there be an access pattern in which $(a_1 \cdots a_T)^N$ is followed by $(b_1 \cdots b_T)^N$. We analyze the behavior of this pattern for a fully associative cache that contains space for storing $K (K < T)$ lines. We assume that the parameter ϵ in BIP is small, and that both sequences in the access pattern repeat many times ($N >> T$ and $N >> K/\epsilon$). Table 3 compares the hit-rate of LRU, OPT, LIP, and BIP for this access pattern.

Table 3: Hit Rate for LRU, OPT, LIP, and BIP

	$(a_1 \cdots a_T)^N$	$(b_1 \cdots b_T)^N$
LRU	0	0
OPT	$(K-1)/T$	$(K-1)/T$
LIP	$(K-1)/T$	0
BIP	$(K-1-\epsilon \cdot [T-K])/T$ $\approx (K-1)/T$	$\approx (K-1-\epsilon \cdot [T-K])/T$ $\approx (K-1)/T$

As the cache size is less than T, LRU causes thrashing and results in zero hits for both sequences. The optimal policy is to retain any $(K-1)$ lines out of the T lines of the cyclic reference so that those $(K-1)$ lines receive hits. After the cache is warmed up, Belady's OPT retains $(K-1)$ blocks out of the T blocks, achieving a hit-rate of $(K-1)/T$ for both sequences. LIP behaves similar to OPT for the first sequence. However, LIP never allows any element of the second sequence to enter the non-LRU position of the cache, thus, causing zero hits for the second sequence.

In each iteration, BIP inserts approximately $\epsilon \cdot (T-K)$ lines in the MRU position which means a hit-rate of $(K-1-\epsilon \cdot [T-K])/T$. As the value of ϵ is small, BIP obtains a hit-rate of approximately $(K-1)/T$, which is similar to the hit-rate of LIP. However, BIP probabilistically allows the lines of any sequence to enter the MRU position. Therefore, when the sequence changes from the first to the second, all the lines in the cache belong to the second sequence after K/ϵ misses. For large N, the transition time from the first sequence to the second sequence is small, and the hit-rate of BIP is approximately equal to $(K-1)/T$. Thus, for small values of ϵ, BIP can respond to changes in the working set while retaining the thrashing protection of LIP.

4.2 Case Studies of Thrashing Workloads

We analyze LIP and BIP in detail using three memory-intensive benchmarks: mcf, art, and health. These benchmarks incur the highest MPKI for the SPEC INT, SPEC FP, and Olden benchmark suite respectively. The LRU policy results in thrashing as the working set of these benchmarks is greater than the baseline 1MB cache. For all experiments in this section a value of $\epsilon = 1/32$ is used.

4.2.1 The mcf benchmark:

Figure 2 shows the code structure from the `implicit.c` file of the mcf benchmark with the three load instructions that are responsible for 84% of the total L2 misses for the baseline cache.

The kernel of mcf can be approximated as linked-list traversals of a data structure whose size is approximately 3.5MB. Figure 3 shows the MPKI for mcf when the cache size is varied under the LRU policy. The MPKI reduces only marginally till 3.5MB and

```
while (arcin)
{
   tail =arcin->tail;
   if(tail->time + arcin->org_cost > latest)
   {
      arcin= (arc_t *) tail->mark;
      continue;
   }
   ...
   arcin= (arc_t *) tail->mark
}
```

Causes 84% of all L2 misses (28% by each instruction)

Figure 2: Miss-causing instructions from the mcf benchmark

then the first *"knee"* of the MPKI curve occurs. LRU results in thrashing for the baseline 1MB cache and almost all the inserted lines are evicted before they can be reused. Both LIP and BIP retain around 1MB out of the 3.5MB working set resulting in hits for at least that fraction of the working set. For the baseline 1MB cache, LRU incurs an MPKI of 136, both LIP and BIP incur an MPKI of 115 (17% reduction over LRU), and OPT incurs an MPKI of 101 (26% reduction over LRU). Thus, both LIP and BIP bridge two-thirds of the gap between LRU and OPT without extra storage.

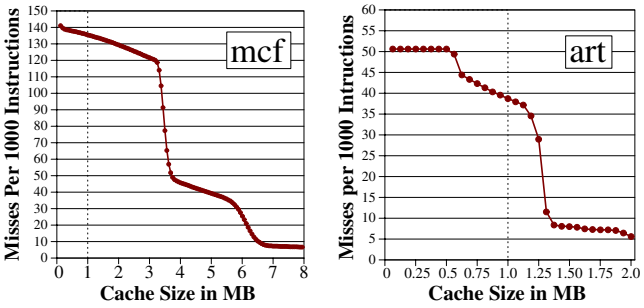


Figure 3: MPKI vs. cache size for mcf and art

4.2.2 *The art benchmark:*

Figure 4 shows the code snippet from the `scanner.c` file of the art benchmark containing the two load instructions that are responsible for 80% of all the misses for the baseline cache. The first load instruction traverses an array of type `f1_layer`. The class of `f1_layer` defines it as a `neuron` containing seven elements of type `double` and one element of type pointer to `double`. Thus, the size of each each object of type `f1_layer` is 64B. For ref-1 input set, `numf1s=10000`, therefore, the total size of the array of `f1_layer` is $64B * 10K = 640KB$. The second load instruction traverses a two dimensional array of type `double`. The total size of this array is equal to $8B * 11 * 10K = 880KB$. Thus, the size of the working set of the kernel of art is approximately 1.5MB.

Figure 3 shows the MPKI of art for varying cache size under LRU replacement. LRU is oblivious to the *"knee"* around 1.5MB and causes thrashing for the baseline 1MB cache. Both LIP and BIP prevent thrashing by retaining a significant fraction of the working set in the cache. For the baseline 1MB cache, LRU incurs an MPKI of 38.7, LIP incurs an MPKI of 23.6 (39% reduction over LRU), BIP incurs an MPKI of 18 (54% reduction over LRU), and OPT incurs an MPKI of 12.8 (67% reduction over LRU). Both LIP and BIP are closer to OPT. The access to array `bus[ti][tj]` brings in cache lines that are not used in later iterations. LIP retains these lines in the cache while BIP can evict these lines. Hence, significantly better MPKI with BIP compared to LIP.

```
numf1s = lwidth*lheight; // = 100*100 for ref input set
numf2s = numObjects+1;   // = 10+1 for ref input set
 •••
for (tj=spot;tj<numf2s;tj++)
{
   Y[tj].y = 0;
   if( !Y[tj].reset )
   for (ti=0;ti<numf1s;ti++)
   Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Causes 41% of all L2 misses

Causes 39% of all L2 misses

Figure 4: Miss-causing instructions from the art benchmark

4.2.3 *The health benchmark:*

Figure 5 shows a code snippet from the `health.c` file. It contains the pointer de-referencing load instruction that is responsible for more than 70% of the misses for the baseline cache.

```
while (list != NULL) {
   ...
   p = list->patient ;
   ...
  list = list->forward;
}
```

Causes 71% of all L2 misses

Figure 5: Miss-causing instruction from the health benchmark

The health benchmark can be approximated as a micro kernel that performs linked list traversals with frequent insertions and deletions. The size of the linked-list data structure increases dynamically with program execution. Thus, the memory reference stream can be approximated as a cyclic reference sequence for which the period increases with time. To show the dynamic change in the size of the working set, we split the benchmark execution into four parts (of approximately 50M instructions each). Figure 6 shows the MPKI of each of the four phases of execution of health as the cache size is varied under the LRU policy.

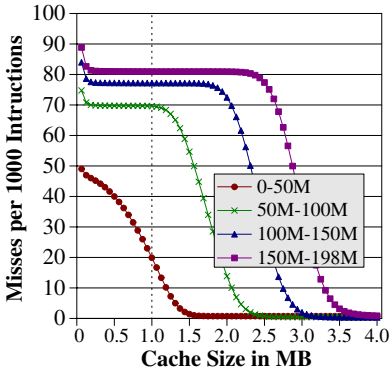


Figure 6: MPKI vs. cache size for health

During the first phase, the size of the working set is less than the baseline 1MB cache so LRU works well. However, in the other three phases, the size of the working set is greater than 1MB, which causes thrashing with LRU. For the full execution of health, LRU incurs an MPKI of 61.7, LIP incurs an MPKI of 38 (38.5% reduction over LRU), BIP incurs an MPKI of 39.5 (36% reduction over LRU), and OPT incurs an MPKI of 34 (45% reduction over LRU).

4.3 Case Study of LRU-Friendly Workload

For workloads that cause thrashing with LRU, both LIP and BIP reduce cache misses significantly. However, some workloads inherently favor the traditional policy of inserting the incoming line at the MRU position. In such cases, changing the insertion policy can hurt cache performance. An example of such a workload is the swim benchmark from the SPEC FP suite. Swim performs matrix multiplies in its kernel. The first *"knee"* of the matrix multiplication occurs at $\frac{1}{2}$ MB while the second *"knee"* occurs at a cache size greater than 64 MB. Figure 7 shows the MPKI for swim as the cache size is increased from $\frac{1}{8}$ MB to 64 MB under LRU policy.

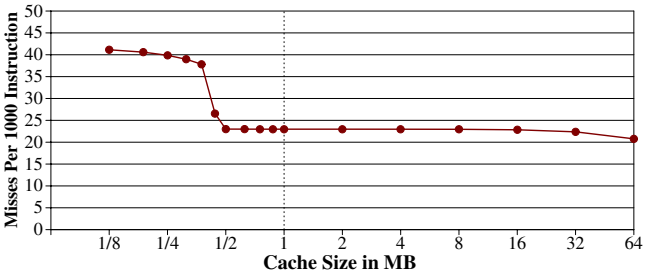


Figure 7: MPKI vs. cache size for swim (x-axis in log scale)

There is a huge reduction in MPKI as the cache size is increased from $\frac{1}{8}$ MB to $\frac{1}{2}$ MB. However, subsequent increase in cache size till 64 MB does not have a significant impact on MPKI. For the baseline cache, the MPKI with both LRU and OPT are similar indicating that there is no scope for reducing misses over the LRU policy. In fact, changes to the insertion policy can only reduce the hits obtained from the middle of the LRU stack for the baseline 1 MB cache. Therefore, both LIP and BIP increase MPKI significantly over the LRU policy. For the baseline cache, LRU incurs an MPKI of 23, LIP incurs an MPKI of 46.5, BIP incurs an MPKI of 44.3, and OPT incurs an MPKI of 22.8.

4.4 Results

Figure 8 shows the reduction in MPKI with the two proposed insertion policies, LIP and BIP, over the baseline LRU replacement policy. For BIP, we show results for $\epsilon = 1/64$, $\epsilon = 1/32$, and $\epsilon = 1/16$ which mean every 64th, 32nd, or 16th miss is inserted in the MRU position respectively.[2]

The thrashing protection of LIP and BIP reduces MPKI by 10% or more for nine out of the sixteen benchmarks. BIP has better MPKI reduction than LIP for art and ammp because it can adapt to changes in the working set of the application. For most applications that benefit from BIP, the amount of benefit is not sensitive to the value of ϵ. For benchmarks equake, parser, bzip2 and swim both LIP and BIP increase the MPKI considerably. This occurs because these workloads either have an LRU friendly access pattern, or the knee of the MPKI curve is less than the cache size and there is no significant benefit from increasing the cache size. For the insertion policy to be useful for a wide variety of workloads, we need a mechanism that can select between the traditional LRU policy and BIP depending on which incurs fewer misses. The next section describes a cost-effective run-time mechanism to choose between LRU and BIP. For the remainder of the paper we use a value of $\epsilon = 1/32$ for all experiments with BIP.

[2]In our studies, we restrict the value of ϵ to 1/power-of-two. To implement BIP, a pseudo-random number generator is required. If there is no pseudo-random number available then an n-bit free running counter can be used to implement a 1-out-of-2^n policy ($n = log_2(1/\epsilon)$). The n-bit counter is incremented on every cache miss. BIP inserts the incoming line in the MRU position only if the value of this n-bit counter is zero. We experimented with both gnu c rand function as well as the 1-out-of-2^n policy and found the results to be similar. Throughout the paper, we report the results for BIP using the 1-out-of-2^n policy assuming that a pseudo-random number generator is not available on chip.

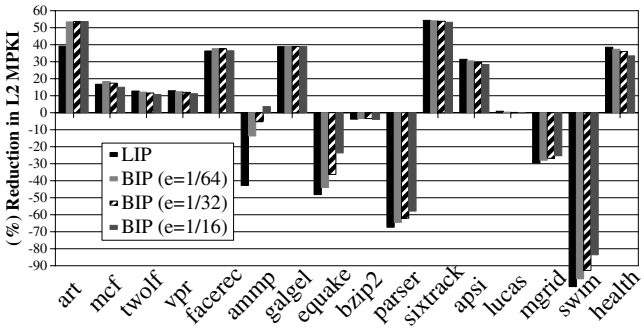


Figure 8: Comparison of Static Insertion Policies

5. DYNAMIC INSERTION POLICY

For some applications BIP has fewer misses than LRU and for some LRU has fewer misses than BIP. We want a mechanism that can choose the insertion policy that has the fewest misses for the application. We propose a mechanism that dynamically estimates the number of misses incurred by the two competing insertion policies and selects the policy that incurs the fewest misses. We call this mechanism *Dynamic Insertion Policy (DIP)*. A straightforward method of implementing DIP is to implement both LRU and BIP in two extra tag directories (data lines are not required to estimate the misses incurred by an insertion policy) and keep track of which of the two policies is doing better. The main tag directory of the cache can then use the policy that incurs the fewest misses. Since this implementation of DIP gathers information globally for all the sets, and enforces a uniform policy for all the sets, we call it *DIP-Global*.

5.1 The DIP-Global Mechanism

Figure 9 demonstrates the working of DIP-Global for a cache containing sixteen sets. Let MTD be the main tag directory of the cache. The two competing policies, LRU and BIP, are each implemented in a separate Auxiliary Tag Directory (ATD). ATD-LRU uses the traditional LRU policy and ATD-BIP uses BIP. Both ATD-LRU and ATD-BIP have the same associativity as the MTD. The access stream visible to MTD is also applied to both ATD-LRU and ATD-BIP. A saturating counter, which we call *Policy Selector (PSEL)*, keeps track of which of the two ATDs incurs fewer misses. All operations on PSEL are done using saturating arithmetic. A miss in ATD-LRU increments PSEL and a miss in ATD-BIP decrements PSEL. The Most Significant Bit (MSB) of PSEL is an indicator of which of the two policies incurs fewer misses. If MSB of PSEL is 1, MTD uses BIP, otherwise MTD uses LRU.

5.2 Implementing DIP via Set Dueling

The DIP-Global mechanism requires a substantial hardware overhead of two extra tag directories. The hardware overhead of comparing two policies can be significantly reduced by using the *Dynamic Set Sampling (DSS)* concept proposed in [13]. The key insight in DSS is that the cache behavior can be approximated with a high probability by sampling few sets in the cache. Thus, DSS

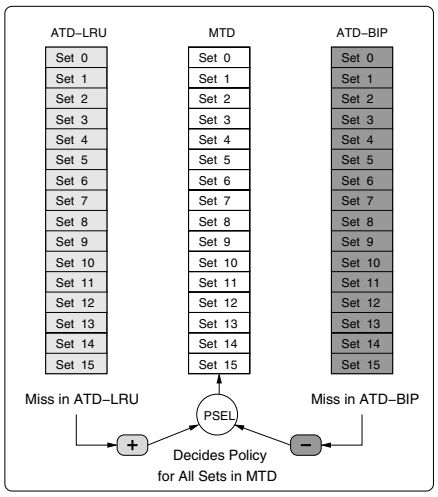


Figure 9: DIP-Global

can significantly reduce the number of ATD entries in DIP-Global from thousand(s) of sets to about 32 sets.

Although DSS significantly reduces the storage required in implementing the ATD (to around 2kB), it still requires building the separate ATD structure. Thus, implementing DIP will still incur the design, verification, and testing overhead of building the separate ATD structure. We propose *Set Dueling*, which obviates the need for a separate ATD structure. The Set Dueling mechanism dedicates few sets of the cache to each of the two competing policies. The policy that incurs fewer misses on the *dedicated sets* is used for the remaining *follower sets*. An implementation of DIP that uses Set Dueling is called *DIP-SD*.

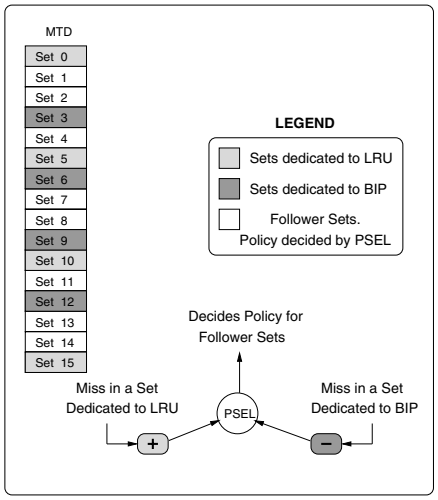


Figure 10: DIP via Set Dueling

Figure 10 demonstrates the working of DIP-SD on a cache containing sixteen sets. Sets 0, 5, 10, and 15 are dedicated to the LRU policy, and Sets 3, 6, 9, and 12 are dedicated to the BIP policy. The remaining sets are follower sets. A miss incurred in the sets dedicated to LRU increments PSEL, whereas, a miss incurred in the sets dedicated to BIP decrements PSEL. If the MSB of PSEL is 0, the follower sets use the LRU policy; otherwise the follower sets use BIP. Note that Set Dueling does not require any separate storage structure other than a single saturating counter.

DIP-SD compares the number of misses across different sets for two competing policies. However, the number of misses incurred by even a single policy varies across different sets in the cache. A natural question is how does the per-set variation in misses of the component policies affect the dynamic selection of Set Dueling? Also, how many dedicated sets are required for DIP-SD to approximate DIP-Global with a high probability? In Appendix A, we derive analytical bounds[3] for DIP-SD as a function of both the number of dedicated sets and the per-set variation in misses of the component policies. The analytical model shows that as few as 32-64 dedicated sets are sufficient for Set Dueling to select between LIP and BIP with a high probability. In Section 5.4 we compare the misses incurred by DIP-SD and DIP-Global.

5.3 Dedicated Set Selection Policy

The dedicated set for each of the competing policies can be selected statically at design time or dynamically at runtime. In this section we describe our method of selecting the dedicated sets. Let N be the number of sets in the cache and K be the number of sets dedicated to each policy (in our studies we restrict the number of dedicated sets to powers of 2). We logically divide the cache into K equally-sized regions each containing N/K sets. Each such region is called a *constituency* [13]. One set is dedicated from each constituency to each of the competing policies. Two bits associated with each set can then identify the set as either a follower set or a dedicated set to one of two competing policies.

We employ a dedicated set selection policy that obviates the need for marking the leader set in each constituency on a per-set basis. We call this policy the *complement-select* policy. For a cache with N sets, the set index consists of $log_2(N)$ bits out of which the most significant $log_2(K)$ bits identify the constituency and the remaining $log_2(N/K)$ bits identify the *offset* from the first set in the constituency. The complement-select policy dedicates to LRU all the sets for which the constituency identifying bits are equal to the offset bits. Similarly, it dedicates to BIP all the sets for which the complement of the offset equals the constituency identifying bits. Thus for the baseline cache with 1024 sets, if 32 sets are to be dedicated to both LRU and BIP, then complement-select dedicates set 0 and every 33rd set to LRU, and Set 31 and every 31st set to BIP. The sets dedicated to LRU can be identified using a five bit comparator for the bits [4:0] to bits [9:5] of the set index. Similarly, the sets dedicated to BIP can be identified using another five bit comparator that compares the complement of bits [4:0] of the set index to bits [9:5] of the set index. Unless stated otherwise, the default implementation of DIP is DIP-SD with 32 dedicated sets using the complement-select policy[4] and a 10-bit[5] PSEL counter.

[3]In [13] a Bernoulli model is used to derive the bounds for DSS. However, as they used an ATD, they were comparing the two policies by implementing both policies for a few sampled sets. Their analytical model does not consider the per-set variation in misses incurred by the component policies. However, in our case, Set Dueling compares the component policies by implementing them on different sets in the cache. Therefore, the analytical model for Set Dueling must take into account the per-set variation in misses incurred by the component policies. Therefore, the bounds derived in [13] are not directly applicable to Set Dueling.

[4]We also experimented with a rand-dynamic policy which randomly dedicates one set from each constituency to each of the two policies LRU and BIP. We invoke rand-dynamic once every 5M retired instructions. The MPKI with both rand-dynamic and complement-select are similar. However, rand-dynamic incurs the hardware overhead of bits for identifying the dedicated sets which are not required for complement-select.

[5]For experiments of DIP-SD in which 64 sets are dedicated to each policy, we use a 11-bit PSEL counter.

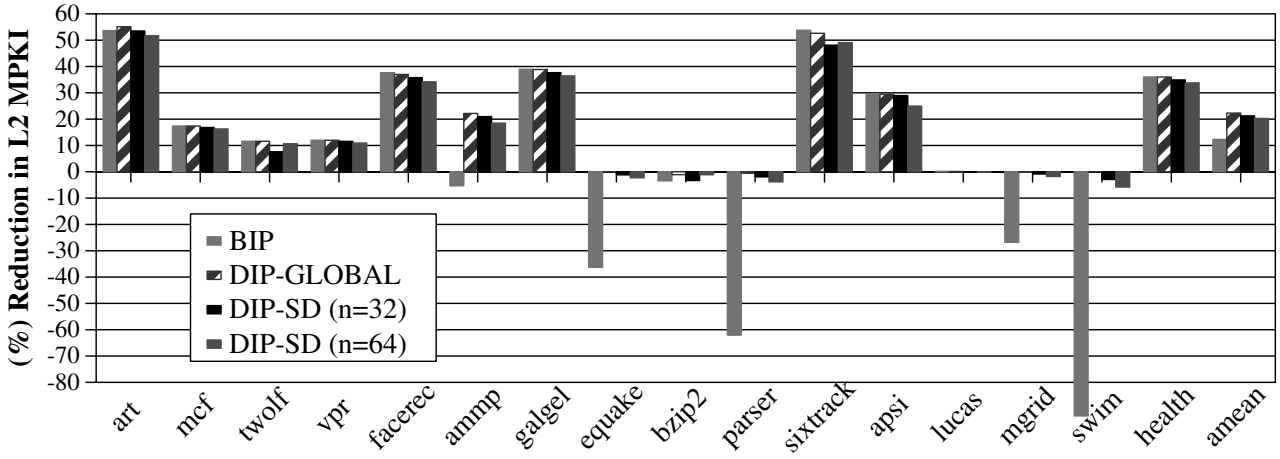


Figure 11: Comparison of Insertion Policies

5.4 Results

Figure 11 shows reduction in MPKI with BIP, DIP-Global, and DIP-SD with 32 and 64 dedicated sets. The bar labeled *amean* is the reduction in arithmetic mean MPKI measured over all the sixteen benchmarks. DIP-Global retains the MPKI reduction of BIP while eliminating the significant MPKI increase of BIP on benchmarks equake, parser, mgrid, and swim. With DIP-Global, no benchmark incurs an MPKI increase of more than 2% over LRU. However, DIP-Global requires a significant hardware overhead of about 128kB. DIP-SD obviates this hardware overhead while obtaining an MPKI reduction similar to DIP-Global for all benchmarks, except twolf. As the number of dedicated sets increases from 32 to 64, the probability of selecting the best policy increases, therefore DIP-SD with 64 dedicated sets behaves similar to DIP-Global for twolf. However, having a large number of dedicated sets also means that a higher fraction (n/N) of sets always use BIP, even if BIP increases MPKI. This causes the MPKI of swim to increase by 5% with 64 dedicated sets. For ammp, DIP reduces MPKI by 20% even though BIP increases MPKI. This happens because in one phase LRU has fewer misses and in the other phase BIP has fewer misses. With DIP, the cache uses the policy best suited to each phase and hence a better MPKI than each of the component policies. We discuss the dynamic adaptation of DIP in more detail in Section 5.5. On average, DIP-Global reduces average MPKI by 22.3%, DIP-SD (with 32 dedicated set) reduces average MPKI by 21.3%, and DIP-SD (with 64 dedicated set) reduces average MPKI by 20.3%.

5.5 Adaptation of DIP to Application

DIP can adapt to different applications as well as different phases of the same application. DIP uses the PSEL counter to select between the component policies. For a 10-bit PSEL counter, a value of 512 or more indicates that DIP uses BIP, otherwise DIP uses LRU. Figure 12 shows the value of the 10-bit PSEL counter over the course of execution for the benchmarks mcf, art, health, swim, and ammp. We sample the value of the PSEL counter once every 1M instructions. The horizontal axis denotes the number of instructions retired (in millions) and the vertical axis represents the value of the PSEL counter.

For mcf, the DIP mechanism almost always uses BIP. For health, the working set during the initial part of the program execution fits in the baseline cache and either policy works well. However, as the dataset increases during program execution, it exceeds the size of

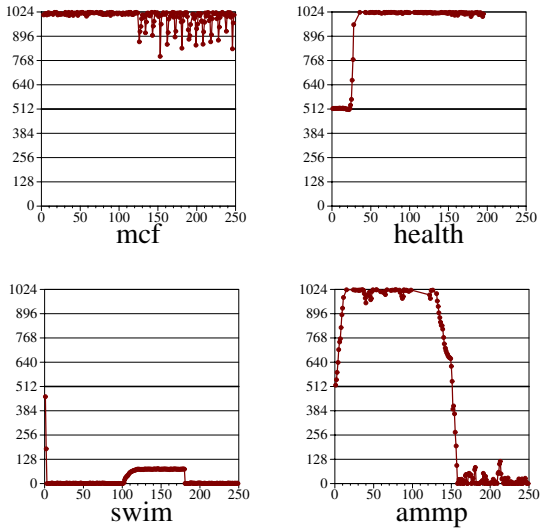


Figure 12: PSEL value during benchmark execution (horizontal axis denotes the number of instruction in Millions)

the baseline cache and LRU causes thrashing. As BIP would have fewer misses than LRU, the PSEL value reaches toward positive saturation and DIP selects BIP. For the LRU friendly benchmark swim, the PSEL value is almost always towards negative saturation, so DIP selects LRU. Ammp has two phases of execution: in the first phase LRU is better and in the second phase BIP is better. With DIP, the policy best suited to each phase is selected; therefore, DIP has better MPKI than either of the component policies standalone.

6. ANALYSIS

6.1 Varying the Cache Size

We vary the cache size from 1 MB to 8 MB and keep the associativity constant at 16-way. Figure 13 shows the MPKI of both LRU and DIP for four cache sizes: 1MB, 2MB, 4MB, and 8MB. The MPKI values are shown relative to the baseline 1MB LRU-managed cache. The bar labeled *avg* represents the arithmetic mean MPKI measured over all the sixteen benchmarks. As mcf has high MPKI, the average without mcf, *avgNomcf*, is also shown.

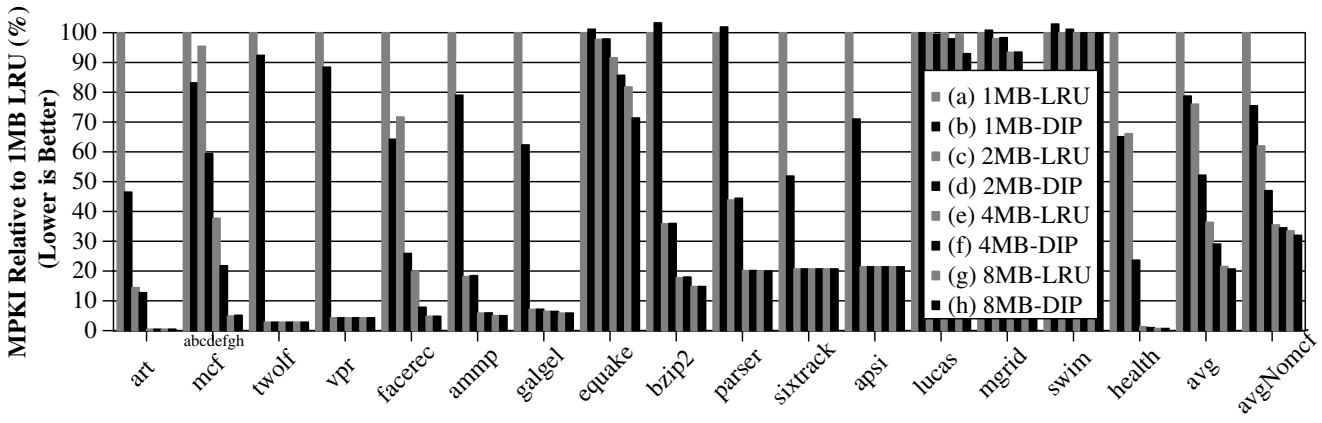


Figure 13: Comparison of LRU and DIP for different cache size

DIP reduces MPKI more than doubling the size of the baseline 1MB cache for benchmarks mcf, facerec, and health. DIP continues to reduce misses for most benchmarks that benefit from increased capacity. The working set of some benchmarks, e.g. vpr and twolf, fits in a 2MB cache. Therefore, neither LRU nor DIP reduces MPKI when the cache size is increased. Overall, DIP significantly reduces average MPKI over LRU for cache sizes up to 4MB.

6.2 Bypassing Instead of Inserting at LRU

DIP uses BIP which inserts most of the incoming lines in the LRU position. If such a line is accessed in the LRU position, only then is it updated to the MRU position. Another reasonable design point is to bypass the incoming line instead of inserting it in the LRU position. A DIP policy that employs BIP which bypasses the incoming line when the incoming line is to be placed in the LRU position is called *DIP-Bypass*. Figure 14 shows the MPKI reduction of DIP and DIP-Bypass over the baseline LRU policy.

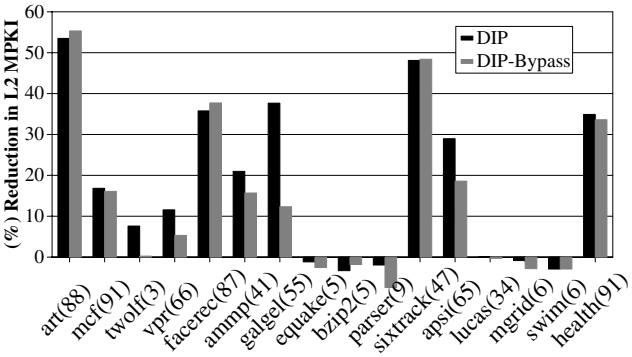


Figure 14: Effect of Bypassing on DIP (The number associated with benchmark shows the percentage of misses bypassed).

For all benchmarks, except art, facerec, and sixtrack, DIP reduces MPKI more than DIP-Bypass. This happens because DIP promotes the line installed in the LRU position to the MRU position if the line is reused, thus increasing the useful lines in the non-LRU positions. On the other hand, DIP-Bypass has the advantage of power savings as it avoids the operation of inserting the line in the cache. The percentage of misses that are bypassed by DIP-Bypass are shown in Figure 14 by a number associated with each benchmark name. Thus, the proposed insertion policies can be used to reduce misses, cache power or both.

6.3 Impact on System Performance

To evaluate the effect of DIP on the overall processor performance, we use an in-house execution-driven simulator based on the Alpha ISA. The relevant parameters of our model are given in Table 5. The processor we model is a four-wide machine with out-of-order execution. Store misses do not block the instruction window unless the 128-entry store buffer is full. The baseline system contains a 1MB 16-way L2 cache which uses LRU replacement. Write-backs from L1 to L2 do not update the replacement information in L2.

Table 4: Baseline system configuration

Machine width	4 instructions/cycle, 4 functional units
Inst. window size	32 instructions
Branch predictor	Hybrid 64k-entry gshare, 64k-entry PAs misprediction penalty is 10 cycles min.
L1 inst. cache	16kB, 64B linesize, 2-way with LRU repl.
L1 data cache	16kB, 64B linesize, 2-way, 2 cycle hit
L2 unified cache	1MB, 64B linesize, 16-way, 6 cycle hit 128-entry store buffer.
Main memory	32 banks, 270 cycle bank access
Off-chip bus	Proc. to bus speed ratio 4:1; 8B/bus-cycle

Figure 15 shows the performance improvement measured in instructions per cycle (IPC) between the baseline system and the same system with DIP. The bar labeled *gmean* is the geometric mean of the individual IPC improvements seen by each benchmark. The system with DIP outperforms the baseline by an average of 9.3%. DIP increases the IPC of benchmarks art, mcf, facerec, and health by more than 15%.

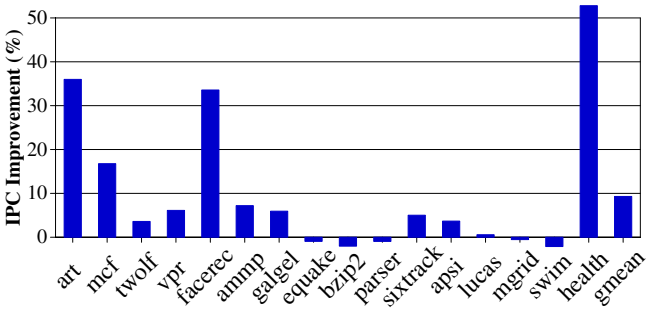


Figure 15: IPC improvement with DIP

6.4 Hardware Overhead and Design Changes

The proposed insertion policies (LIP, BIP, and DIP) require negligible hardware overhead and design changes. LIP inserts all incoming lines in the LRU position, which can easily be implemented by not performing the update to the MRU position that occurs on cache insertion.[6] BIP is similar to LIP, except that it infrequently inserts an incoming line into the MRU position. To control the rate of MRU insertion in BIP, we use a five-bit counter (BIPCTR). BIPCTR is incremented on every cache miss. BIP inserts the incoming line in the MRU position only if the BIPCTR is zero. Thus, BIP incurs a storage overhead of 5 bits. DIP requires storage for the 10-bit saturating counter (PSEL). The complement-select policy avoids extra storage for identifying the dedicated sets.

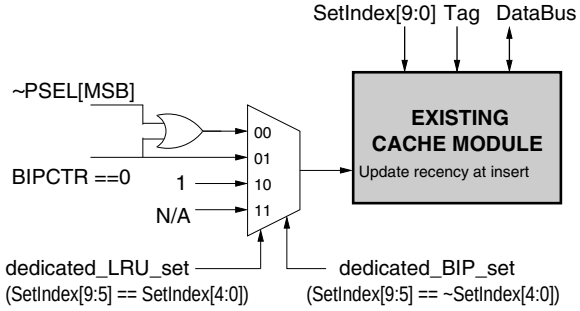


Figure 16: Hardware changes for implementing DIP

Figure 16 shows the design changes incurred in implementing DIP. The implementation requires a total storage overhead of 15 bits (5-bit BIPCTR + 10-bit PSEL) and negligible logic overhead. A particularly attractive aspect of DIP is that it does not require extra bits in the tag-store entry, thus avoiding changes to the existing structure of the cache. The absence of extra structures also means that DIP does not incur power and complexity overheads. As DIP does not add any logic to the cache access path, the access time of the cache remains unaffected.

7. RELATED WORK

Cache replacement studies have received much attention from both industry and academia. We summarize the work that most closely relates to the techniques proposed in this paper, distinguishing our work where appropriate.

7.1 Alternative Cache Replacement Policies

The problem of thrashing can be mitigated with replacement schemes that are resistant to thrashing. If the working set of an application is only slightly greater than the available cache size, then even a naive scheme such as random replacement can have fewer misses than LRU. For the baseline cache random replacement reduces MPKI for the thrashing workloads: art by 34%, mcf by 1.6%, facerec by 14.4%, and health by 16.9%, whereas, DIP reduces MPKI for art by 54%, mcf by 17%, facerec by 36% and health by 35%. Thus, the effectiveness of random replacement at reducing misses significantly reduces as the size of the working set increases. In [4], an analytical model that captures and explains the difference in performance of various cache replacement policies is studied. Several studies [15][5][14] have looked at including frequency (reuse count) information for improving cache replacement.

[6]LIP, BIP, and DIP do not rely on true LRU which makes them amenable to the LRU approximations widely used in current on-chip caches.

7.2 Hybrid Replacement

For workloads that cause thrashing with LRU, both random-based and frequency-based replacement schemes have fewer misses than LRU. However, these schemes significantly increase the misses for LRU-friendly workloads. Recent studies have investigated hybrid replacement schemes that dynamically select from two or more competing replacement policies. Examples of hybrid replacement schemes include *Sampling-Based Adaptive Replacement (SBAR)* [13] and *Adaptive Cache (AC)* [18]. The problem with hybrid replacement is that it may require tracking separate replacement information for each of the competing policies. For example, if the two policies are LRU and LFU (Least Frequently Used), then each tag-entry in the baseline cache needs to be appended with frequency counters ($\geq$ 5-bits each) which must be updated on each access. Also, the dynamic selection requires extra structures (2kB for SBAR and 34kB for AC) which consume hardware and power. Table 5 compares SBAR-based hybrid replacement[7] between LRU and the following schemes: *MRU-Repl* replaces the MRU line, *NMRU-mid* [4] replaces a line randomly from the less recent half of the recency stack, *Rand* is random replacement, *RLRU-Skew (RMRU-Skew)* [4] is a skewed random policy that uses a linearly increasing (decreasing) replacement probability for recency positions ranging from MRU to LRU, and *LFU* is the least frequently used policy implemented using five-bit saturating counters [18]. DIP outperforms the best performing hybrid-replacement while obviating the design changes, hardware overhead, power overhead, and complexity of hybrid replacement. In fact, DIP bridges two-third of the gap between LRU and OPT while requiring less than two bytes of extra storage.

Table 5: Comparison of replacement policies

Replacement Policy	%Reduction in MPKI over LRU	Hardware Overhead
SBAR (LRU + MRU-Repl)	8.8	2 kB
SBAR (LRU + NMRU-mid)	5.1	2 kB
SBAR (LRU + Rand)	8.9	2 kB
SBAR (LRU + RLRU-Skew)	6.6	2 kB
SBAR (LRU + RMRU-Skew)	11.3	2 kB
SBAR (LRU + LFU)	14.7	12 kB
DIP	**21.3**	**2 B**
Belady's OPT	32.2	N/A

7.3 Related Work in Paging Domain

We also analyze some of the related replacement studies from the paging domain. Early Eviction LRU (EELRU) [17] tracks the hits obtained from each recency position for a larger sized cache. If there are significantly more hits from the recency positions larger than the cache size, EELRU changes the eviction point of the resident pages. For the studies reported in [17], EELRU tracked 2.5 times as many pages as in physical memory. We analyzed EELRU for our workloads with 2.5 times the tag-store entries. EELRU reduces the average MPKI by 13.8% compared to DIP which reduces average MPKI by 21.3%.

A recent proposal, Adaptive Replacement Cache (ARC) [11], maintains two lists: *recency list* and *frequency list*. The recency list contains pages that were touched only once while resident, whereas the frequency list contains pages that were touched at least twice. ARC dynamically tunes the number of pages devoted to each list. We simulated ARC for our workloads and found that ARC reduces average MPKI by 5.64% and requires 64kB storage.

[7]The MPKI reduction provided by SBAR and AC are similar [18].

7.4 Cache Bypassing and Early Eviction

Several studies have investigated cache bypassing and early eviction. McFarling[10] proposed dynamic exclusion to reduce conflict misses in a direct-mapped instruction cache. Gonzalez et al. [3] proposed using a *locality prediction table* to bypass access patterns that are likely to pollute the cache. Johnson [6] used reuse counters with a *macro address table* to bypass lines with low reuse. Several proposals [19] [21][20] exist for bypassing or early eviction of lines brought by instructions with low locality. Another area of research has been to predict the last touch to a cache line [8] [9]. After the predicted last touch, the line can either be turned off [7] or be used to store prefetched data [8].

However, *locality*, *liveness* and *last touch* are a function of both the replacement policy and the available cache size. For example, if a cyclic reference pattern with a working set size slightly greater than the available cache size is applied to a LRU-managed cache, all the inserted lines will have poor locality, will be dead as soon as they are installed, and will have their last touch at insertion. The solution in such a case is neither to bypass all the lines nor to evict them early, but to retain some fraction of the working set so that it provides cache hits. DIP retains some fraction of the working set for longer than LRU, thus obtaining hits for at least those lines.

8. CONCLUSIONS AND FUTURE WORK

The commonly used LRU replacement policy performs poorly for memory-intensive workloads that reuse a working set greater than the available cache size. The LRU policy inserts a line and evicts it before it is likely to be reused causing a majority of the lines in the cache to have zero reuse. In such cases, retaining some fraction of the working set would provide hits for at least that fraction of the working set. This paper separates the problem of replacement into two parts: *victim selection policy* and *insertion policy*. Victim selection deals with which line gets evicted to install the incoming line. The insertion policy deals with where on the replacement stack the incoming line is placed when installing it in the cache. We show that simple changes to the insertion policy can significantly improve the cache performance of memory-intensive workloads, and make the following contributions:

1. We propose the LRU Insertion Policy (LIP) which inserts all the incoming lines in the LRU position. We show that LIP can protect against thrashing and yields close to optimal hit-rate for applications with a cyclic reference pattern.

2. We propose the Bimodal Insertion Policy (BIP) as an enhancement to LIP that allows for aging and adapting to changes in the working set of an application. BIP infrequently inserts an incoming line in the MRU position, which allows it to respond to changes in the working set while retaining the thrashing protection of LIP.

3. We propose a Dynamic Insertion Policy (DIP) that dynamically chooses between BIP and traditional LRU replacement. DIP uses BIP for workloads that benefit from BIP while retaining traditional LRU for workloads that are LRU-friendly and incur increased misses with BIP.

4. We propose Set Dueling to implement cost-effective dynamic selection between competing policies. Set Dueling dedicates a small percentage of sets in the cache to each of the two component policies and chooses the policy that has fewer misses on the dedicated set for the remaining follower sets. Set Dueling does not require any additional storage, except for a single saturating counter.

We show that DIP reduces the average MPKI of a 1MB 16-way L2 cache by 21%, while incurring less than two bytes of storage overhead and almost no change to the existing cache structure.

Although this study evaluated the cache performance of workloads on a uni-processor system, it provides insights that can easily be extended to shared caches in a multi-core system. Set Dueling can be also be applied for cost-effective implementation of optimizations such as choosing between different cache management policies, dynamically tuning prefetchers, or detecting phase changes. Exploring these extensions is a part of our future work.

9. ACKNOWLEDGMENTS

Thanks to Archna Monga for discussions on random sampling. The feedback from Andy Glew and William Hasenplaugh greatly helped in improving the quality of this work. We thank Veynu Narasiman and Aater Suleman for their comments and feedback. The studies at UT were supported by gifts from IBM, Intel, and the Cockrell Foundation.

10. REFERENCES

[1] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. In *IBM Systems journal*, pages 78–101, 1966.

[2] W. fen Lin et al. Reducing dram latencies with an integrated memory hierarchy design. In *HPCA-7*, pages 301–312, 2001.

[3] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS-9*, 1995.

[4] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. *SIGMETRICS Perform. Eval. Rev.*, 34(1):228–239, 2006.

[5] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *ISCA-27*, 2000.

[6] T. L. Johnson. *Run-time adaptive cache management*. PhD thesis, University of Illinois, Urbana, IL, May 1998.

[7] S. Kaxiras et al. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA-28*, 2001.

[8] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA-28*, 2001.

[9] W. Lin and S. Reinhardt. Predicting last-touch references under optimal replacement. In *Technical Report CSE-TR-447-02, University of Michigan*, 2002.

[10] S. McFarling. Cache replacement with dynamic exclusion. In *ISCA-19*, pages 191–200, 1992.

[11] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceeding of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[12] E. Perelman et al. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.

[13] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA-33*, 2006.

[14] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *ISCA-32*, pages 544–555, 2005.

[15] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS '90*, 1990.

[16] S. Ross. *A First Course in Probability*. Pearson Prentice Hall, 2006.

[17] Y. Smaragdakis et al. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, 2003.

[18] R. Subramanian et al. Adaptive caches: Effective shaping of cache behavior to workloads. In *MICRO-39*, 2006.

[19] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO-28*, 1995.

[20] Z. Wang et al. Using the compiler to improve cache replacement decisions. In *PACT*, page 199, 2002.

[21] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *HPCA-6*, 2000.

Appendix A: Analytical Model for Set Dueling

Let there be N sets in the cache. Let Set Dueling be used to choose between two policies $P1$ and $P2$. When policy P1 is implemented on all the sets in the cache, the average number of misses per set is μ_1 with standard deviation σ_1. Similarly, when policy P2 is implemented on all the sets in the cache, the average number of misses per set is μ_2 with standard deviation σ_2. Let Δ denote the difference in average misses $|\mu_1 - \mu_2|$ and σ denote the combined standard deviation $\sqrt{\sigma_1^2 + \sigma_2^2}$.

Let n sets be randomly selected from the cache to estimate the misses with policy P1 and another group of n sets be randomly selected to estimate the misses with policy P2. We assume that the number of dedicated sets n is sufficiently large such that by the *central limit theorem* [16] the sampling distribution can be approximated as a Gaussian distribution. We also assume that n is sufficiently small compared to the total number of sets in the cache (N) so that removing the n sets does not significantly change the mean and standard deviation of the remaining $(N - n)$ sets. To derive the bounds for Set Dueling we use the following well-established results [16] for sampling distribution: *If the distribution of two independent random variables have the means μ_a and μ_b and the standard deviation σ_a and σ_b, then the distribution of their sum (or difference) has the mean $\mu_a + \mu_b$ (or $\mu_a - \mu_b$) and the standard deviation $\sqrt{\sigma_a^2 + \sigma_b^2}$.*

Let $sum1$ be the total number of misses for the n sets dedicated to policy P1. Then, by central limit theorem, $sum1$ can be approximated as a Gaussian random variable with mean μ_{sum1} and standard deviation σ_{sum1}, given by:

$$\mu_{sum1} = n \cdot \mu_1 \tag{1}$$

$$\sigma_{sum1} = \sqrt{\sum \sigma_1^2} = \sqrt{n} \cdot \sigma_1, \tag{2}$$

Similarly, let $sum2$ be the total number of misses for the n sets dedicated to policy P2. Then, $sum2$ can also be approximated as a Gaussian random variable with mean μ_{sum2} and standard deviation σ_{sum2}, given by:

$$\mu_{sum2} = n \cdot \mu_2 \tag{3}$$

$$\sigma_{sum2} = \sqrt{\sum \sigma_2^2} = \sqrt{n} \cdot \sigma_2, \tag{4}$$

The PSEL counter tracks the difference in $sum1$ and $sum2$ and selects the policy that has fewer misses on the sampled sets. Let θ be the difference in value of the two sums, i.e. $\theta = sum1 - sum2$. Because $sum1$ and $sum2$ are Gaussian random variables, θ is also a Gaussian random variable with mean μ_θ and standard deviation σ_θ given by:

$$\mu_\theta = \mu_{sum1} - \mu_{sum2} = n \cdot \mu_1 - n \cdot \mu_2 = n \cdot (\mu_1 - \mu_2) \tag{5}$$

$$\sigma_\theta = \sqrt{\sigma_{sum1}^2 + \sigma_{sum2}^2} = \sqrt{n \cdot \sigma_1^2 + n \cdot \sigma_2^2} \tag{6}$$

$$= \sqrt{n} \cdot \sqrt{\sigma_1^2 + \sigma_2^2} = \sqrt{n} \cdot \sigma, \tag{7}$$

$where\ \sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$

Let policy P2 have fewer misses than policy P1, i.e. $\mu_1 > \mu_2$. Then, for Set Dueling to select the best policy, $\theta > 0$. If $P(Best)$ is the probability that Set Dueling selects the best policy, then $P(Best)$ can be written as:

$$P(Best) = P(\theta > 0) = P(\frac{(\theta - \mu_\theta)}{\sigma_\theta} > \frac{(0 - \mu_\theta)}{\sigma_\theta}) \tag{8}$$

$$P(Best) = P(Z > \frac{-n \cdot (\mu_1 - \mu_2)}{\sqrt{n} \cdot \sigma}), \tag{9}$$

$where\ Z = \frac{(\theta - \mu_\theta)}{\sigma_\theta}\ is\ the\ standard\ Gaussian\ variable$

$$P(Best) = 1 - P(Z > \frac{n \cdot (\mu_1 - \mu_2)}{\sqrt{n} \cdot \sigma}) \tag{10}$$

$as\ P(Z > -z) = 1 - P(Z > z)$

$$P(Best) = 1 - P(Z > \sqrt{n} \cdot \frac{\Delta}{\sigma}),\quad where\ \Delta = |\mu_1 - \mu_2|\ ,\ \mu_1 > \mu_2 \tag{11}$$

$$P(Best) = 1 - P(Z > \sqrt{n} \cdot r),\quad where\quad r = \frac{\Delta}{\sigma} \tag{12}$$

Z is the standard Gaussian variable for which the value $P(Z > z)$ can be obtained using standard statistical tables. Equation 12 can be used to compute P(Best) for any two policies. For example, if for policy P1, $\mu_1 = 100$ and $\sigma_1 = 12$ and for policy P2, $\mu_2 = 94$ and $\sigma_2 = 16$. Then, $\Delta = 6$, $\sigma = 20$ and $r = 0.3$. For n=32, $P(Best) = 1 - P(Z > \sqrt{32} \cdot 0.3) = 1 - P(Z > 1.7) = 96\%$.

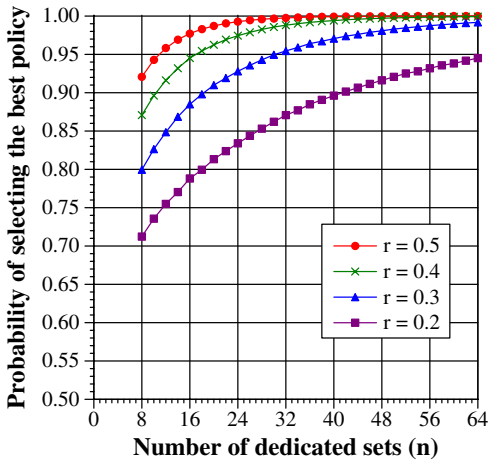


Figure 17: Analytical Bounds for Set Dueling

Figure 17 shows the variation in P(Best) as the number of dedicated sets is changed for different values of the r metric. The r metric is a function of workload, cache organization, and the relative difference between the two policies. For most of the benchmarks studied, the r-metric for the two policies LRU and BIP is more than 0.2 indicating that 32-64 sampled sets are sufficient for Set Dueling to select the best policy with a high probability. Thus, Set Dueling can be implemented by dedicating about 32 to 64 sets to each of the two policies, LRU and BIP, and using the winning policy (of the dedicated sets) for the remaining sets.